Advanced Thread Synchronization for Multithreaded MPI Implementations

Hoang-Vu Dang
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: hdang8@illinois.edu

Sangmin Seo, Abdelhalim Amer, and Pavan Balaji
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439, USA
Emails: {sseo,aamer,balaji}@anl.gov

***Abstract*—Concurrent multithreaded access to the Message Passing Interface (MPI) is gaining importance to support emerging hybrid MPI applications. The interoperability between threads and MPI, however, is complex and renders efficient implementations nontrivial. Prior studies showed that threads waiting for communication progress (waiting threads) often interfere with others (active threads) and degrade their progress. This situation occurs when both classes of threads compete for the same MPI resource and ownership passing to waiting threads does not guarantee communication to advance. The best-known practical solution prioritizes active threads and adapts first-in-first-out arbitration within each class. This approach, however, suffers from residual wasted resource acquisitions (waste) and ignores data locality, thus resulting in poor scalability.**

In this work, we propose thread synchronization improvements to eliminate waste while preserving data locality in a production MPI implementation. First, we leverage MPI knowledge and a fast synchronization method to eliminate waste and accelerate progress. Second, we rely on a cooperative progress model that dynamically elects and restricts a single waiting thread to drive a communication context for improved data locality. Third, we prioritize active threads and synchronize them with a locality-preserving lock that is hierarchical and exploits unbounded bias for high throughput. Results show significant improvement in synthetic microbenchmarks and two MPI+OpenMP applications.

***Index Terms*—MPI; threads; OpenMP; thread safety; lock; mutex; synchronization**

I. Introduction

Message Passing Interface (MPI) applications are moving toward interoperating with MPI through multiple threads. The primary driving factors are ease of programmability for emerging fine-grained threading models and the desire to efficiently utilize modern network fabrics, which require multiple communicating cores to fully exploit their capabilities. In order to meet such expectations, thread safety is a prerequisite, and its corresponding overheads should be minimal.

The intricacies of the MPI standard render designing and implementing correct and efficient support for threading nontrivial. As a result, most production implementations satisfy the core of the thread compliance through locks, since using exclusively lock-free or wait-free objects is complex to implement and to maintain. MPI implementations were shown to suffer significantly from scalability issues due to lock contention. Thus, several works explored different directions for potential improvements. Three orthogonal aspects contribute to the costs of locking: *lock granularity*, *ownership passing latency*, and *ownership arbitration*. Lock granularity in MPI implementations were explored in prior works [1], [2], and reducing lock ownership passing latency has been extensively studied [3], [4], [5]. In our prior work [6], we identified *arbitration* as an important factor that was missing from the literature. Here, we build on our prior findings and provide further insight and improvements that combine more advanced arbitration and locality-preserving methods.

The major arbitration aspect of locking in MPI implementations is the relation between threads waiting for communication progress (*waiting threads*), which occurs in routines with blocking semantics, and the others (*active threads*). When threads in either of these states compete for the same resource (e.g., a critical section that protects a message queue), ownership passing to a waiting thread does not guarantee communication to advance. Consequently, wasted resource acquisitions unnecessarily degrade the progress of active threads. We showed in our prior work that such a scenario often occurs when an unfair lock is monopolized by waiting threads [6]. We also demonstrated that prioritizing active threads and adopting first-in-first-out (FIFO) ownership passing within each class of threads can significantly improve communication progress. This method, however, scales poorly. First, it suffers from residual wasted acquisitions caused by a blind $O(N)$ lookup complexity to find a thread capable of making progress among N waiters. Second, it is data locality oblivious.

We propose in this work to build on our prioritization method a synchronization model that incorporates MPI and hardware information to achieve an $O(1)$ reactivation of waiting threads and locality preservation. The key components of this method are as follows. First, at most one thread among waiters (none if there are no waiters), called the *server*, is elected to drive a communication context. Restricting access to a single waiter improves locality of the communication context data structures. Second, the reactivation of waiting threads is driven by MPI knowledge; waiting threads use private *synchronization counters* to wait, track their pending MPI operations, and receive wakeup signals from the server on their completion. Third, in addition to raising the priority of active threads, we synchronize their concurrent accesses with a locality-preserving high-throughput lock achieved by a

combination of unbounded lock monopolization and NUMA-awareness. Results show significant improvement in synthetic microbenchmarks and two MPI+OpenMP applications.

II. Thread Safety in MPI Implementations

Supporting `MPI_THREAD_MULTIPLE`—allowing multiple threads to concurrently call MPI—is nontrivial. First, the MPI standard imposes certain ordering[1] and progress[2] guarantees on thread-compliant implementations, which restrict the thread safety design space. Second, application threads are allowed to share MPI objects, such as communicators and requests, and may suffer contention because of the resulting sharing of MPI internal data structures (e.g., message queues). Combined with the large set of MPI routines with disparate thread safety requirements, these intricacies make correct and efficient support for threading arduous. As a result, most production MPI runtimes implement the core of the thread compliance through locking, since lock-free or wait-free designs are complex to implement and to maintain. Moreover, the majority of MPI implementations ensure thread safety through a single global lock (often referred to as *global* granularity). Such a conservative approach is prone to high contention, but its simplicity makes correct threading support less error prone than with fine-grained designs. In this study, we also focus on the global granularity. Other related work on the topic of thread granularity is reviewed in Section VI.

For a lock-based MPI implementation, *arbitration* of the concurrent accesses is an important factor. Suppose that one thread is performing a blocking operation (e.g., `MPI_Recv` or `MPI_Wait`) and successfully acquires the global lock of the MPI library. If the operation cannot be satisfied immediately, however, the thread **must** release the lock. Failure to do so prevents other threads from entering the critical section, violates MPI progress requirements, and may lead to a deadlock. Thus, arbitrating lock ownership has correctness implications. In the remainder of paper, all our thread-safe methods guarantee arbitration correctness, and the focus will be their performance implications.

We illustrate the relationship between thread safety and MPI communication in Figure 1. It describes a simplified implementation of `MPI_Isend` and `MPI_Wait`. These routines are examples of a nonblocking MPI call, to send a message, and a blocking MPI call, to wait for its completion. In the example we distinguish two major code paths with distinct progress properties. The first, which we refer to as the *main path*, is taken by both routines between the first lock acquisition (lines 18 and 40) and the last lock release operation (lines 22 and 44). This code path is similar to most MPI routines and often advances the system. The other path, which we refer to as the *progress loop*, concerns only blocking calls. It is characterized by a tight loop (line 28) waiting for the completion of a target operation. This path does not guarantee progress and is characterized by high-frequency lock acquire/release operations. Thus, by our prior definition, threads executing the progress loop are waiting threads and the others are active threads. We consider lock acquisitions by waiters as wasted (or simply as causing waste) when they yield no progress while active threads are waiting for the lock.

Regardless of the state in which a thread is, however, the arbitration of the concurrent accesses is dictated solely by the lock arbitration. That is, lock ownership passing defines the order in which threads execute the critical section. In order to promote progress of the system, our prior work exploited this distinction between the threads in the locking implementation [6], [7]. We considered active threads as having a higher priority; thus we extended traditional lock acquisition interfaces with a low-priority interface. The goal was to have waiting threads acquire the lock with lower priority than that of active threads (line 33). By combining this method with FIFO locks (ticket [8] and CLH [9]), we achieved better communication progress by promoting operation injection into the network and reducing waste. This solution, however, relies on a blind FIFO arbitration among waiting threads. That is, if only one among N threads has its operation completed, $O(N)$ lock-passing operations are needed in order to reactivate it.

In this paper, we leverage knowledge from the MPI runtime to reactivate waiting threads in $O(1)$ complexity. Furthermore, prior works ignored the memory hierarchy and incurred heavy lock-passing costs. We address this issue as well by promoting locality-preserving practices. We use the $O(N)$ locality-oblivious prior work implemented on top of the FIFO-based CLH lock [9] and the production MPICH [10] as the baseline for comparison. We choose MPICH for its thread-compliance maturity and its relevance as one of the most widely adopted MPI libraries, used directly or indirectly through third-party platform-specific tuned derivatives. We do believe, however, that the findings in this paper can be beneficial to other MPI implementations as well as other communication runtimes.

III. Shortcomings of Existing Methods

This section re-evaluates the issues in the multithreaded communication and describes the motivation for our methods. For clarity, we define the following terminology:

- MTX: MPICH implementation using a Pthreads mutex as a lock (this is the same as the MPICH 3.2 released)
- CLH: MPICH implementation using the CLH lock [7]
- P-CLH: MPICH implementation using the CLH lock with prioritization of the main path [7]

A. Experimental Setup

We use latency, bandwidth, and message rate as metrics to evaluate the performance of our baseline CLH and P-CLH. We note that the performance of the baseline is superior to that of MTX for `MPI_THREAD_MULTIPLE` when there is no oversubscription [7].

The latency benchmark is similar to the multithreaded OSU latency benchmark (`osu_latency_mt` [11]) except that we

[1]When threads issue ordered MPI operations (e.g., point-to-point) and the user establishes a relative ordering among the corresponding threads, such order needs to be preserved.

[2]For instance, threads waiting in MPI blocking calls must not block the progress of other threads.

```
typedef struct request {
  REQUEST_BODY;
  bool complete;
  void *(*completion_cb)(*);
} request_t;

// Global lock
lock_t g_lock;

// Callback to complete a request
void complete_request(request_t *req) {
  req->complete = true;
}

// Issuing a nonblocking send operation
void MPI_Isend(..., MPI_Request *handle) {
  request_t *req;
  acquire(g_lock);
  req = create_request();
  req->completion_cb = complete_request;
  Isend_body(req);
  release(g_lock);
  set_request(handle, req);
}

// Internal progress routine
void progress_wait(request_t *req) {
  while (!req->complete) {
    bool made_progress = poll_network();
    if (!made_progress) {
      release(g_lock);
      yield();
      acquire_low(g_lock);
    }
  }
}

// Waiting for request completion
void MPI_Wait(..., MPI_Request *handle) {
  acquire(g_lock);
  request_t *req = get_request(handle);
  progress_wait(req);
  *handle = MPI_REQUEST_NULL;
  release(g_lock);
}
```

Fig. 1. Simplified thread-safe implementation of `MPI_Isend` and `MPI_Wait`. It assumes a global critical section (protected by `g_lock`) and a callback-based request completion. `poll_network` is a hardware network call that progresses all outstanding operations. In particular, during this call, user-provided completion callback functions are executed when the corresponding operations have completed. `complete_request` simply marks the request as complete. Lock ownership passing is ensured through the `acquire`, `acquire_low` (for low priority), and `release` calls.

used OpenMP instead of Pthreads to take advantage of the OpenMP thread-binding capability. This benchmark uses two MPI ranks, one per compute node. One MPI rank (*sender*) performs a number of pairs (10,000 for messages up to size $\leq$ 8 KB, 1,000 for larger messages) of `MPI_Send` and `MPI_Recv` to the other MPI rank (*receiver*). The receiver creates a fixed number of threads to perform a corresponding `MPI_Recv` and `MPI_Send` such that the total number of requests matches the sender's. The latency is computed at the sender by taking the average time used per message.

The bandwidth and message rate benchmarks are the same as those studied in [6]. These benchmarks also create two MPI ranks in two different nodes (sender and receiver); each creates the same number of OpenMP threads. The sender performs a number of `MPI_Isends` in each thread (the default 64 messages is used in our experiments) before waiting for all of them with `MPI_Waitall`; It then performs `MPI_Recv`. The receiver side similarly performs `MPI_Irecv`, `MPI_Waitall`, and `MPI_Send` in each thread to match the sender. This process is repeated so that the number of messages is the same as that in the latency benchmark. The message rate is computed at the sender as the total number of messages over the overall execution time. The bandwidth is computed similarly but using the total size of transferred data instead of the number of messages.

Our platform for the experiments is a cluster of Intel Haswell-EP machines. Each node consists of two Intel Xeon E5-2699 v3 CPUs (36 cores in total) whose cores are arranged into four NUMA domains. The nodes are interconnected by using Mellanox FDR InfiniBand. We compiled our benchmarks and MPICH (for both CLH and P-CLH) using the Intel compiler 16.0.3 with the MXM low-level communication runtime included in HPC-X 1.6.392. We used the `MPI_T` instrumentation available in MPICH for profiling and instrumentation, in order to obtain breakdowns in timing and internal counters such as the number of network polls, and HPC-Toolkit [12] for measuring the number of cache misses. All our tests were done with each thread bound to a CPU core using two OpenMP environment variables: `OMP_PROC_BIND=close` and `OMP_PLACES=cores`. MPICH uses a single communication context per MPI process; thus we expect high degrees of contention for communication-intensive codes. Furthermore, message rates are bound by single-threaded performance because of the limited concurrency of a globally locked single-communication context implementation.

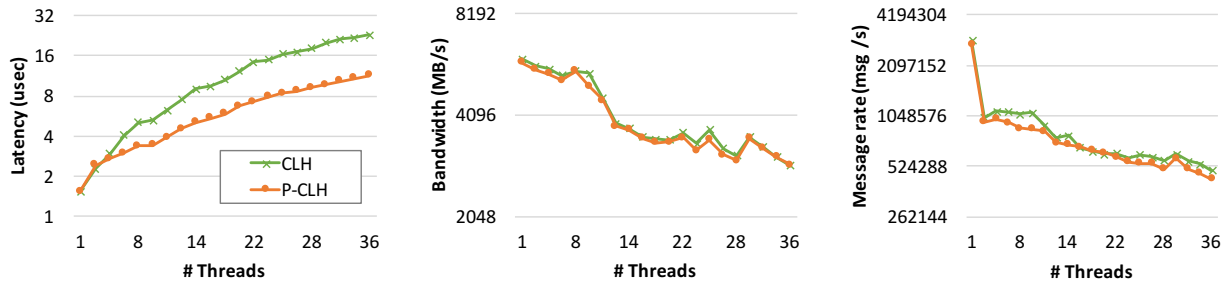
B. Performance Results

Figure 2 illustrates the results of each benchmark. The results show that as the number of threads increases from 1 to 36, the communication performance degrades significantly. For instance, while the latency measured with one thread is about 1.5 μs, it increases to 23 μs for CLH and 11.5 μs for P-CLH with 36 threads. In the following subsections, we analyze the performance of each benchmark in detail.

C. Analysis of Latency and Bandwidth Results

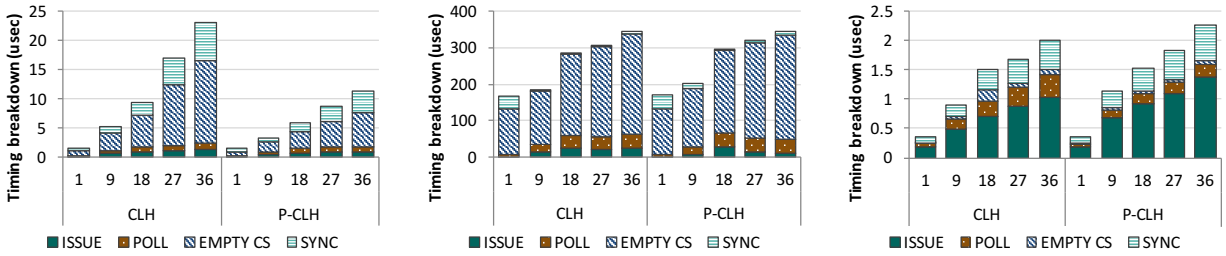
To reveal where most of the execution times in Figure 2 are spent, we divided the execution time into four parts as shown in Figure 3. The breakdown timing is measured and reported at the sender side (similar results can be obtained at the receiver side). In the figure, as the number of threads increases, the execution time of each portion increases. Most notably, in the latency and bandwidth cases, we see a large increase in EMPTY CS, that is, wasted time in the critical section (CS) without doing useful work.

The waste in the latency and bandwidth results is caused primarily by the increased number of network polls when more threads are involved in the communication. Figure 4 shows



(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Fig. 2. Initial performance results of CLH and P-CLH. The message size is 64 bytes for the latency and message rate benchmarks and is 1 MB for the bandwidth benchmark.



(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Fig. 3. Execution time breakdown of the performance results in Figure 2. ISSUE, POLL, EMPTY CS, and SYNC represent time spent in issuing operations, time spent in making progress, time wasted in the CS without doing useful work, and time spent in synchronizations, respectively. The timing overhead is about 8%, 1% and 6% on average (harmonic mean) of the total execution time for the latency, bandwidth and message rate benchmark respectively.

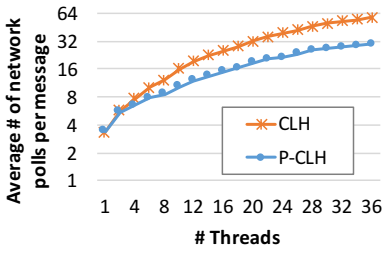


Fig. 4. Average number of network polls performed per message in the latency benchmark with CLH and P-CLH.

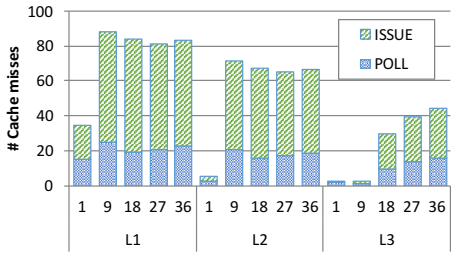


Fig. 5. Average number of L1, L2, and L3 cache misses per message for different numbers of threads in the message rate benchmark using P-CLH. The breakdown represents the numbers of misses that occurred in `MPI_Isend` or `MPI_Irecv` (ISSUE) and in `MPI_Wait` (POLL).

the average number of network polls performed per message in the latency benchmark (we omit the result of the bandwidth benchmark because it showed a similar pattern). Although the number of messages is constant across all experiments, the number of network polls per message increases as the number of threads increases. Recall that any waiting threads executing a blocking operation can enter the CS through the progress loop. If more waiting threads enter the CS and perform unnecessary work (e.g., network polling), it can delay active threads from proceeding and consequently slow the overall performance. By prioritizing the main path, P-CLH enables more work to be injected into the runtime compared with CLH; hence it improves the performance because of the presence of more active threads. However, P-CLH still suffers from $O(N)$ FIFO ownership passing between N waiting threads, which reduces its performance with increased numbers of threads.

D. Analysis of Message Rate Results

The execution time breakdown of the message rate results in Figure 3(c) shows that the majority of its time is spent in issuing messages (i.e., ISSUE). The overall message rate is again confirmed to be bounded by the issuing rate, as also reported in [7]. This phenomenon can be understood better by looking at the number of cache misses in Figure 5. As we increase the number of threads in the experiment, the number of cache misses increases substantially—first in the L1 and L2

```
typedef struct request {
  REQUEST_BODY;
  bool complete;
  void *(*completion_cb)(*);
  scount_t scounter;
} request_t;

bool server = false; // is there a server?
list<scount_t *> waiters; // list of waiters

// Callback when a request is finished
void complete_request(request_t *req) {
  req->complete = true;
  list_remove(waiters, &req->scounter);
  scount_signal(&req->scounter, false);
}

void progress_wait(request_t *req) {
  bool elected = false; // am I the server?
  while (!req->complete) {
    bool made_progress = poll_network();
    if (!made_progress) {
      if (!server) {    // no active server
        elected = true; // I am elected
        server = true;  // as a server
      }
      if (elected) { // I am a server
        release(g_lock);
        yield();
        acquire_low(g_lock);
      } else {       // I am a waiter
        list_append(waiters, &req->scounter);
        scount_wait(&req->scounter,g_lock,1);
      }
    }
  }
  if (elected)      // I am no longer
    server = false; // the server
  // Wake up a potential server
  if (!server && !list_empty(waiters))
    scount_signal(list_pop(waiters), true);
}
```

Fig. 6. Modifications to Figure 1 for selective reactivation.

```
typedef struct scount {
  int    cur_count;
  cond_t cvar;
} scount_t;

/* Wait for N events. This routine assumes
   the lock L associated to the condition
   variable C->cvar is held. */
void scount_wait(scount_t *C, lock_t L,
                 int N) {
  C->cur_count = N;
  /* cond_wait releases L to wait for signal,
     then reacquires it at returns. */
  cond_wait(&C->cvar, L);
}

/* Signal one event or force-wakeup on
   condition variable C->cvar. */
void scount_signal(scount_t *C, bool force) {
  if (force) cond_signal(&C->cvar);
  else {
    C->cur_count--;
    if (!C->cur_count) cond_signal(&C->cvar);
  }
}
```

Fig. 7. Example implementation of a *synchronization counter*. It assumes the existence of generic condition variable (`cvar`) and lock (`L`) implementations. The routine `scount_wait` allows waiting for `N` events and `scount_signal` decrements the number of pending events (`cur_count`) for `C` and wakes up the corresponding thread when there are no more pending events. We also allow forced wakeup with the Boolean `force`.

caches within the same NUMA node ($\leq$ 9 threads) and then in the L3 cache when going out of the NUMA node ($>$ 9 threads). This increasing rate reflects the trend of our message rate benchmark results in Figure 2(c).

The increase in cache misses happens mainly in the issuing part of the benchmark because of the FIFO ownership passing exercised in the CLH lock used in CLH and P-CLH. Even though issuers initiate MPI operations and make progress, they take turns in entering the CS, thus causing the data movement and resulting in the increase in cache misses. The decreasing message rate with a larger number of threads signifies the problem of thread synchronization, especially lock ownership passing, between active threads.

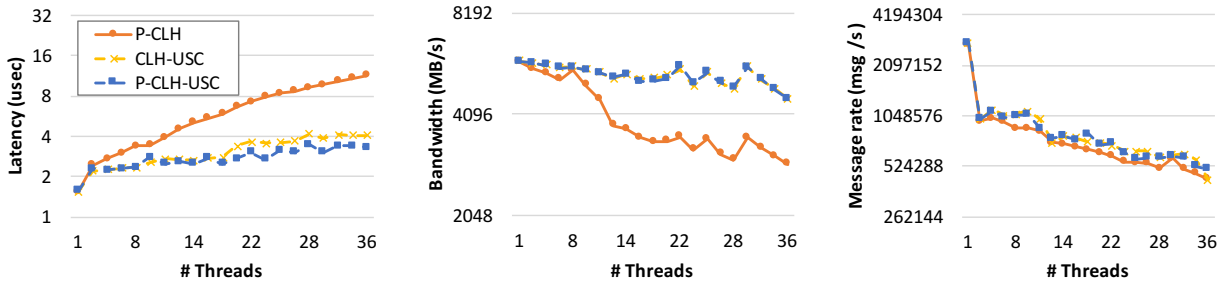
IV. Thread Synchronization Techniques

In this section, we present our solutions for the problems presented in Section III.

A. Selective Reactivation of Waiting Threads

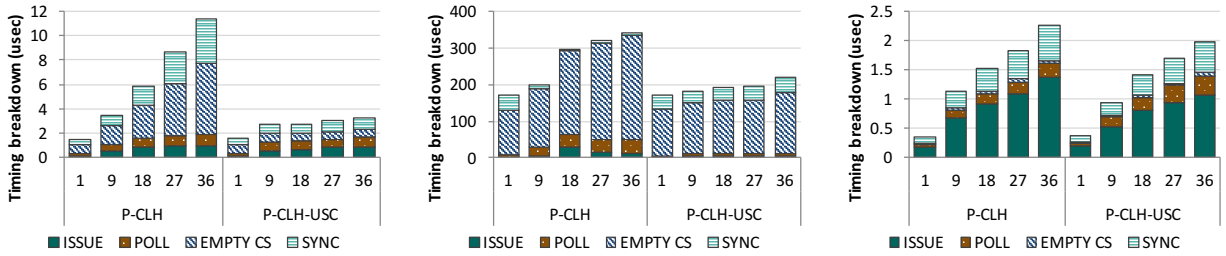
Our idea for solving the latency and bandwidth problems described in Section III-C is to allow only one waiting thread to drive a communication context while making all other waiting threads wait outside the CS until their request is completed. At most one thread, called the *server*, is elected among all waiting threads, and only the server is allowed to enter the CS and poll the network for communication progression. In contrast to a previously studied approach that utilizes a dedicated communication server [13], our method is a decentralized one that can be easily integrated in existing MPI implementations, which do not assume a centralized entity for making progress or thread arbitration. Restricting access to a single waiter also increases the likelihood of residency in cache of the communication context data structures and avoids contention for the CS.

Remaining or new waiting threads become *waiter*s and wait until the server completes their request. A waiter indicates its waiting intent through a *synchronization counter* whose initial value equals the number of pending operations that it is waiting for. When the server completes a request, it enables the waiter associated with the request to continue the execution with a signal (i.e., work-driven, selective reactivation). To avoid starvation, when the server finishes its own request, it elects another waiter, if there is one, to hand over its server role; otherwise, the waiters will be waiting forever, since none is making progress. Note that the number of servers depends on the network hardware and communication volume. For our purpose, a single server is sufficient; but the selective reactivation method can easily be extended for multiple servers. Figure 6 depicts our selective reactivation method with a single server.



(a) Latency (the lower. the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Fig. 8. Performance results of CLH and P-CLH with selective reactivation. The message size is 64 bytes for the latency and message rate benchmarks and is 1 MB for the bandwidth benchmark. CLH-USC and P-CLH-USC represent the results using selective reactivation with a user-level synchronization counter implementation along with CLH and P-CLH, respectively.



(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message rate (the higher, the better)

Fig. 9. Execution time breakdown of the P-CLH and P-CLH-USC results in Figure 8. We omit the breakdown of the CLH-USC results because it shows a pattern similar to that of the P-CLH-USC results. ISSUE, POLL, EMPTY CS, and SYNC represent time spent in issuing operations, time spent in making progress, time wasted in CS without doing useful work, and time spent in synchronizations, respectively. The timing overhead is about 8%, 1%, and 7% on average (harmonic mean) of the total execution time for the latency, bandwidth, and message rate benchmark respectively.

Implementing the selective reactivation requires two important changes to the runtime: (1) defining a synchronization counter implementation and (2) associating a counter per blocking operation and storing its reference in the corresponding request objects. In Figure 6, we assume a generic synchronization counter object referred to by the abstract type `scount_t`. This object supports two types of operation: a signal operation is translated to `scount_signal()`, and a wait operation is translated to `scount_wait()`. Each request contains a `scount_t` object (line 5) representing our synchronization structure. For an `MPI_Waitall` operation, a reference to the counter object will be initialized to the number of pending operations and stored in all of them to consume signal events.

Since a request object is associated with a `scount_t` object, we can tie the waiting thread to the request by making the waiting thread wait on the associated `scount_t` object. In addition, because the one-to-one mapping between a thread and a `scount_t` object is maintained, one `scount_t` object can be regarded as representing one waiting thread. If the thread cannot complete the request and is required to wait, it will not enter the CS using the *progress loop* path but instead will be blocked inside the synchronization object if a server already exists, as shown at line 33. When the request is finished, the signal operation is performed by the server at the callback function (line 15). The `scount_t` objects are linked in a globally shared doubly linked list (line 9), so that the server can find a target waiter when needed. With the doubly linked list, we can efficiently remove a waiter from the list when it is signaled, that is, when the request is finished by the server.

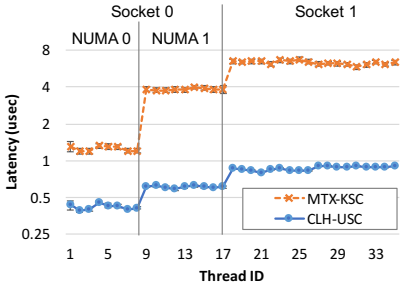
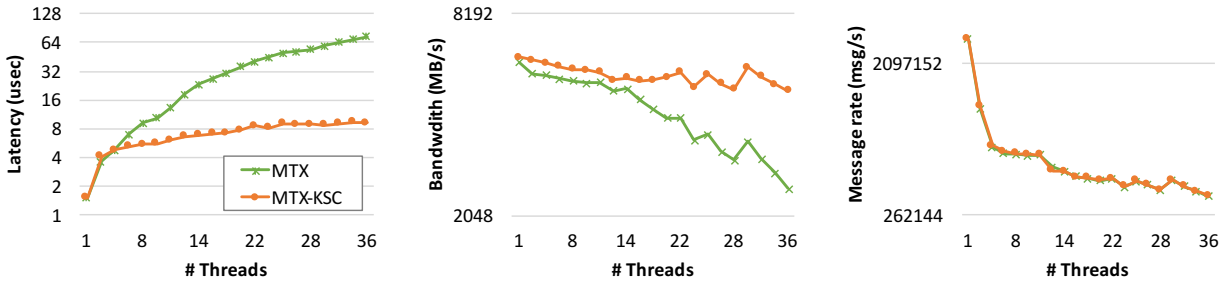


Fig. 11. Ping-pong latency between a thread pinned to core 0 and another thread pinned to a different core indicated by the Thread ID using (1) Pthreads mutex and kernel-level synchronization counter (MTX-KSC) and (2) the CLH lock and user-level synchronization counter (CLH-USC).

An example implementation of the synchronization counter is illustrated in Figure 7. It assumes generic lock and condition



(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Fig. 10. Performance results of the vanilla MPICH-3.2 (MTX) and our modification using selective reactivation with the Pthreads mutex and kernel-level synchronization counter (MTX-KSC). The message size is 64 bytes for the latency and message rate and is 1 MB for the bandwidth benchmark.

variable implementations underneath. A `scount_wait` operation publishes the intent to wait for `N` events. `scount_signal` events decrement the counter and wake up the thread when reaching zero (i.e., all events are satisfied). The `force` argument forces a thread wakeup regardless of the counter value. This is essential for electing the next server when the current one is leaving the runtime. In the case of waiting for the completion of multiple requests, more than one request object may share one synchronization counter object. The server issues signals whenever it completes one of the requests, and a thread wakeup is triggered when the counter has reached 0 (see line 15 in Figure 6 and the routine `scount_signal` in Figure 7). This strategy prevents the waiter thread from being woken up multiple times, many of those just to figure out that not all its requests are finished, and thus adding meaningless synchronization overhead.

To support other blocking operations without requests such as `MPI_Win_flush`, we can tie the waiting thread to associated pending objects, for example, `MPI_Window`, in a similar way. We leave for future work the task of extending our selective reactivation method to support these kinds of operations.

We implemented the selective reactivation method in the baseline CLH and P-CLH. Figures 8 and 9 illustrate the performance results of our implementations with the microbenchmarks and the execution time breakdowns, respectively. The results show that compared with the baseline, our selective reactivation method is effective in lowering the communication latency in multithreaded communication cases (Figure 8(a)) and in improving the bandwidth significantly with a larger number of threads (Figure 8(b)). Specifically, the latency and bandwidth are improved by 3 times and 5 times with 36 threads, respectively. These improvements come mainly from the reduction of the time spent in EMPTY CS, as shown in Figures 9(a) and 9(b).

The selective reactivation method can also be applied to MTX by using a kernel-level synchronization counter, which can be implemented with a Pthreads condition variable and a counter. However, since the kernel-level synchronization counter has higher latency due to its kernel-specific implementation, it should be used only when oversubscription is required. The performance improvement when applying the selective reactivation technique to MTX is shown in Figure 10. The selective reactivation (denoted as MTX-KSC) performs significantly better than the baseline in the latency and bandwidth benchmarks. However, the absolute latency of MTX-KSC is still far worse than that of CLH and P-CLH using selective reactivation (P-CLH-USC and P-CLH-USC in Figure 8(a)) because of the higher latency of both the kernel-level lock and synchronization counter implementations. Figure 11 shows up to sixfold differences in latency when using a kernel-level and a user-level lock and synchronization counter.

B. *Locality-Preserving Locking with Unbounded Bias*

In the preceding subsection, we showed that our selective reactivation technique can eliminate wasted executions and ensure that any thread entering the CS has a high chance of performing useful work. This method, however, does not improve the performance of the message rate case (Figure 9(c)). The reason is that message rates were bound by the injection rate of nonblocking calls to the runtime that suffers from intranode data movement, as was analyzed in Section III-D. Based on that analysis, we propose a new locking strategy that is designed to reduce cache misses.

First, we introduce a "locality-preserving locking with unbounded bias" method based on CLH, which we refer to as CLHub (Figure 12). The lock is designed to provide the necessary property for our purpose—the lock is biased toward the high-priority thread that most recently released the lock. It is implemented by combining a simple spin lock (i.e., a biased lock) to exploit the lock monopolization with two CLH locks to handle high and low priorities while taking advantage of the FIFO lock (e.g., reducing thread contention on the lock by queuing threads in the lock structure). In the figure, the data structure for the lock includes the following fields (lines 1–6): `bias` – a spin lock that has a biased behavior; `fifoH` and `fifoL` – CLH locks to block threads in the high-priority and low-priority paths, respectively; and `filter` – a flag to switch between two priorities.

The lock allows only high-priority threads (i.e., active threads in our case), which call `acquire()`, to utilize the monopolization, while low-priority threads (i.e., waiting threads)

```
1  typedef struct clhub {
2    spin_lock_t bias; // biased lock
3    clh_t fifoH; // FIFO lock for high priority
4    clh_t fifoL; // FIFO lock for low priority
5    int filter;  // to switch between different
                     priority paths
6  } *clhub_t;
7
8  void acquire(clhub_t l) {
9    if (try_acquire(l->bias) == fail) {
10     acquire(l->fifoH);
11     l->filter = 1;
12     acquire(l->bias);
13     l->filter = 0;
14     release(l->fifoH);
15   }
16 }
17
18 void acquire_low(clhub_t l) {
19   acquire(l->fifoL);
20   while (l->filter == 1) {
21   }; // busy wait.
22   acquire(l->bias);
23   release(l->fifoL);
24 }
25
26 void release(clhub_t l) {
27   release(l->bias);
28 }
```

Fig. 12. Pseudocode of the locality-preserving lock with unbounded bias, which uses a combination of a spin lock and two CLH locks. For simplicity, we use the same function names (`acquire` and `release`) for different lock types, but please consider them as overloaded functions that can handle proper lock types.

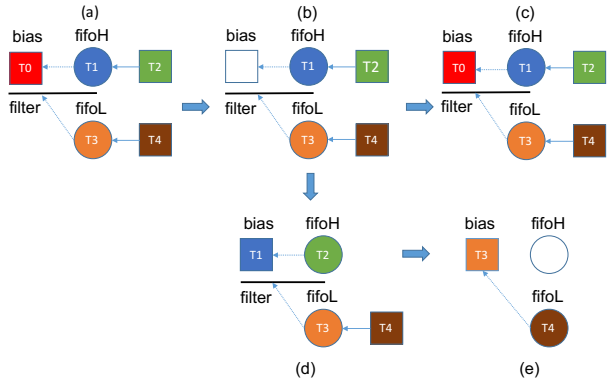


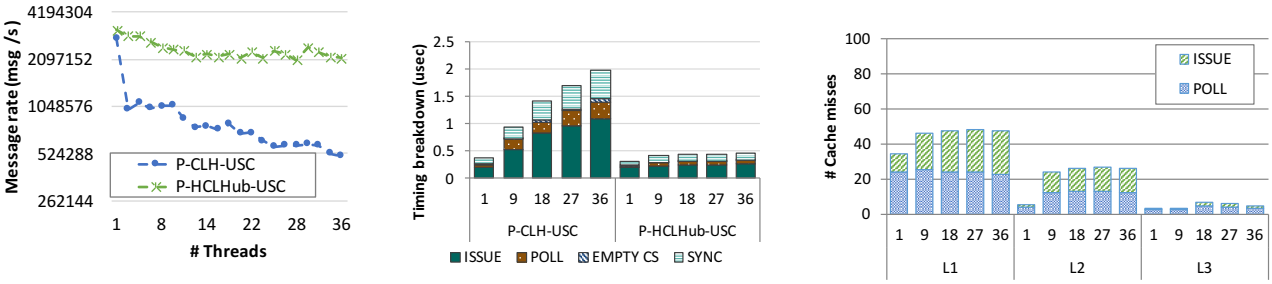
Fig. 13. Illustration of five threads (T0,..,T4) using the locality-preserving lock: (a) T0 performs `acquire()` and succeeds, taking `bias`. Right after T0, T1 and T2 perform `acquire()` while T3 and T4 perform `acquire_low()`. As a result, T1, T2, T3, and T4 spin at `bias` (i.e., T1 becomes *candidate*), `fifoH`, `filter`, and `fifoL`, respectively. (b) T0 releases `bias`. (c) If T0 calls `acquire()` immediately again, it has a higher chance of taking `bias` than does T1 because of locality. (d) If T0 finishes and moves on, T1 can succeed, acquiring `bias`. (e) Only when there is no high-priority thread is `filter` released and low-priority threads can acquire `bias`.

rely on the FIFO property of the lock without the monopolization through `acquire_low()`. Since active threads always advance the system, in addition to raising their priority over waiting threads, we synchronize their concurrent accesses with a locality-preserving high-throughput lock. The locality preservation is achieved through a competitive ownership passing, which results in core-level unbounded lock monopolization. The monopolization achieves locality preservation of the lock and critical section data but does not cause starvation for waiting threads in practice since active threads are guaranteed to complete their operations in a bounded number of steps.

Figure 13 illustrates an example usage of our locality-preserving lock of Figure 12. In the figure, threads T0, T1, and T2 perform `acquire()`, and T0 succeeds initially. In `acquire()`, trying to acquire the biased lock (`bias`) first (line 9 in Figure 12) allows the same thread, here T0, to execute the CS in a loop without interfering with other threads because of the lock monopolization behavior. Both T1 and T2 fail to acquire `bias`, and thus they attempt to acquire a FIFO lock, `fifoH` (line 10). Only T1 will succeed and become the *candidate* for entering the CS after T0; T2 is queued in `fifoH`. The candidate T1 waits on the biased lock for its turn (line 12) but is able to succeed only if T0 releases and does not immediately reacquire the lock. When that happens, T1 becomes the owner of `bias`, and it elects T2 waiting on `fifoH` as the candidate by releasing `fifoH` (line 14). On the other hand, when `acquire_low()` is used (as for T3 and T4), the FIFO behavior is maintained as threads will be queued up in `fifoL` (line 19). `filter` serves as a mechanism to switch between low-priority and high-priority paths when low and high priorities are mixed. The low-priority thread runs only when there is not a concurrently running high-priority thread.

Now, we combine our locality-preserving lock with a NUMA-aware lock using the "lock cohorting" technique [3]. The technique uses a hierarchical locking strategy (i.e., two levels of lock, one at each NUMA node and the other at global scope) in order to allow prioritizing ownership passing to threads in the same NUMA node and to make it NUMA-aware with minimal cost. Since fairness is not needed at the high-priority branch, we replace `fifoH` in Figure 12 with this NUMA-aware lock to further improve cache locality. Note that this hierarchical locality-preserving lock results in NUMA-node level unbounded lock monopolization. We implemented this lock using a spin lock for the global scope and a CLH lock for the NUMA node, which is similar to C-BO-MCS described in [3] except that we replace MCS with our existing CLH lock implementation and we do not employ back-off for the spinlock.

Figure 14 presents our optimization results for the message rate. We note that our hierarchical locality-preserving lock method does not affect the latency and bandwidth because the latency benchmark is using blocking calls and the bandwidth is limited by memory rather than cache. Although some performance loss still occurs, we are able to obtain a message rate of 2 million messages per second with 36 threads. This improvement is due to the significant reduction in the number of cache misses across all caches, as shown in Figure 14(c), compared with that shown in Figure 5. We believe that our performance number is the best message rate recorded for this benchmark with this large number of kernel threads. We



(a) Message rate for 64-byte message.

(b) Execution time breakdown.

(c) Breakdown of cache misses occurred in `MPI_Isend` or `MPI_Irecv` (ISSUE) and in `MPI_Wait` (POLL).

Fig. 14. Results of using the selective reactivation technique in combination with the P-CLH lock (denoted as P-CLH-USC) and our hierarchical locality-preserving lock (denoted as P-HCLHub-USC). The timing overhead is about 8% on average (harmonic mean) of the total execution time in the timing breakdown analysis.

achieve a fivefold performance improvement compared with the performance of the baseline case.

V. Evaluation

All the experiments in this section were performed on the Stampede cluster [14] each node of which is equipped with dual 8-core Intel Xeon E5-2680 processors (16 cores in total) and 32 GB of memory. Although the number of cores is less than that of the previous cluster used for running the benchmarks (which is less ideal for our cases), Stampede allows us to experiment with more machine nodes while using the same network interface (an FDR Mellanox device). We compiled our programs with the Intel compiler version 15.0.2 using `-O3`. The MPI implementations were built with the default configuration (`-O2`), and the MXM layer for the MPI implementation was the same as that described in Section III-A. All our tests were done with each thread bound to a CPU core using two OpenMP environment variables: `OMP_PROC_BIND=close` and `OMP_PLACES=cores`.

We first re-evaluated our communication benchmarks on Stampede. Then, to further analyze the behavior of our methods in practice, we ported two miniapps to use `MPI_THREAD_MULTIPLE` and for multiple threads to perform communications. In each experiment, we compared two MPI implementations: one using a priority-based FIFO lock (P-CLH) and the other with our two techniques presented in Section IV (P-HCLHub-USC).

A. Communication Benchmarks

Figure 15 shows performance results of the communication benchmarks that we used in the preceding sections. The results on Stampede are also consistent with our previous results obtained on Haswell clusters. Our method outperforms the baseline for all tested cases. The improvement of our method ranges from 1.4× to 3.5× depending on the message size.

B. Graph500

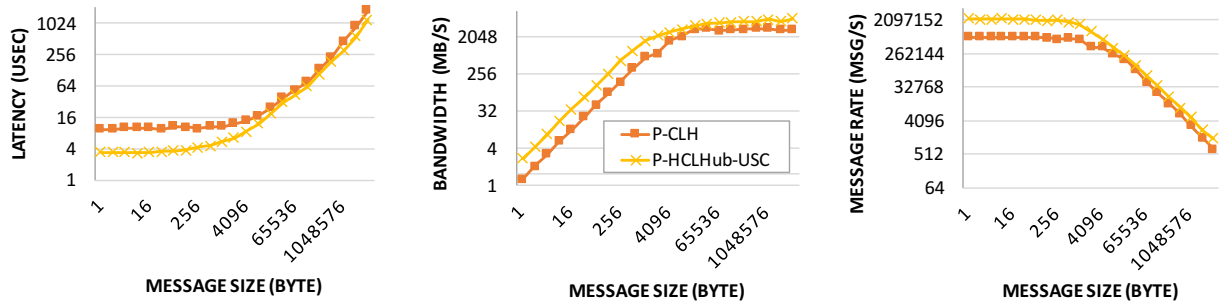
Graph500 [15] is a communication kernel that generates a large-scale graph, assigns to each MPI process a fixed set of vertices, and cooperatively traverses the graph in a breadth-first manner until all vertices are visited. The kernel represents irregular access patterns and fine-grained communication. It is frequently used to evaluate the communication layer of programming models and runtime systems.

Our reference implementation is obtained from the hybrid approach implementation described in [16]. We modified this implementation by converting nonblocking calls to blocking calls and dedicating a subset of threads (half in this experiment) to perform message receiving. We designate half the threads, whose ID is an odd number, as senders and the other half, whose ID is an even number, as receivers so that the data locality is improved. With this version, we more effectively evaluate our improvement with the selective reactivation technique since blocking calls are used. We also find that this implementation achieves better performance than that reported before in the literature because of the poor performance of MPI blocking calls inside threads.

Performance results of the Graph500 miniapp for both the reference implementation and our modification are illustrated in Figure 16. As the results indicate, MPI runtime enhanced with our techniques (P-HCLHub-USC) consistently outperforms the baseline (P-CLH). We achieve 13% improvement using the reference implementation and 2× improvement (harmonic mean) using our modification in terms of TEPS. Since this miniapp is communication-bound and all threads heavily participate in communication, how to manage thread synchronization and arbitration is a major factor for achieving better performance. In this regard, P-HCLHub-USC significantly improves the communication latency in the multithreaded communication case compared with the baseline.

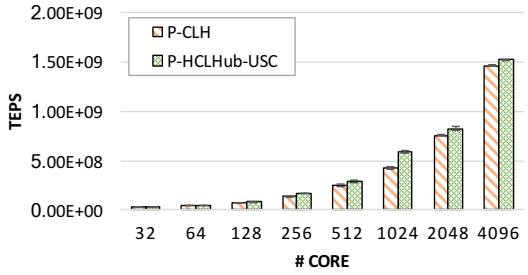
C. HPCCG

HPCCG is a miniapp from the Mantevo benchmark suite [17]. The miniapp represents a close approximation to a finite-volume application. The communication pattern is irregular, mainly due to several sparse matrix-vector multiplication steps generated by the application. The problem size is determined by the size of the matrix, in this case generated by the number of rows per process. The communication is performed prior to the local computation by using mainly point-to-point MPI calls.

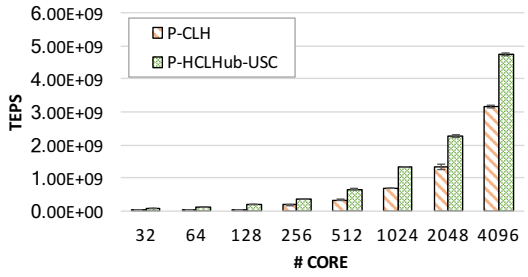


(a) Latency (the lower, the better) (b) Bandwidth (the higher, the better) (c) Message Rate (the higher, the better)

Fig. 15. Performance results on Stampede in terms of latency, bandwidth, and message rate for 16 threads with variable message sizes.



(a) Results of Graph500 using non-blocking calls



(b) Results of Graph500 using blocking calls

Fig. 16. Graph500 results (harmonic mean over 16 runs) using a weak-scaling experiment with problem size scale 24 per node (i.e., 32 at 256 nodes) and 16 threads per node, in terms of traversed edges per second (TEPS – the higher, the better).

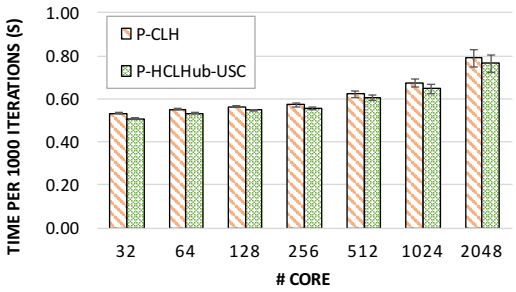


Fig. 17. Performance results of HPCCG using weak scaling with 128K matrix rows per node and 16 threads per node (the lower, the better).

The benchmark suite also provides three implementations: MPI-only, MPI+OpenMP, and OpenMP. The MPI+OpenMP implementation, however, is a `MPI_THREAD_SINGLE` application where OpenMP is used only for parallel loops. Our strategy for the hybrid implementation is to further subdivide the matrix into smaller domains and assign those smaller domains to each thread. Thus, each thread has to perform both communication and computation. Communication between threads in a node is done via shared memory for collectives and shared states with appropriate synchronization.

The performance results of HPCCG are shown in Figure 17. The MPI runtime incorporated with our methods (P-HCLHub-USC) was able to outperform the baseline (P-CLH) by 3–5% reduction in the execution time due to 20–25% improvement in point-to-point communication. The overall performance improvement is small, however, and statistically significant only up to 64 nodes because of the dominance of the computation and allreduce operation, which is not improved by our techniques (becomes >95% of the overall execution time at 128 nodes). From this experiment we confirm that when the application is computation-bound, our techniques do not hurt its performance; indeed, even in this case our methods can improve the performance, as long as the application contains situations where threads contend for the communication.

VI. Related Work

Some early work on supporting the interoperability between threads and MPI, such as MiMPI [18] and MPICH-MT [19], focused only on thread safety issues, since multicore machines did not exist at that time. Another approach is to implement MPI processes as threads [20], [21], [22], [23], [24]. While this approach can bring performance benefits for on-node communication by exploiting efficient data sharing between threads, it requires completely new implementations of both the MPI runtime and shared-memory programming model runtime. It also needs compiler support to privatize global variables for each thread because MPI processes, which are implemented as threads, have to own separate memory space for global variables.

The issue of granularity and arbitration in supporting thread safety has been studied before. For example, Dózsa et al. [2]

and Balaji et al. [1] studied the replacement of the MPI coarse-grained lock with fine-grained locks and implemented parallel receive queues using these locks. The implementation proved to be complex and error-prone, however, and thus was not completed. On the other hand, thread arbitration was studied in detail first in [6], [7] and showed significant improvements. In this paper, we have generalized the previous techniques and improved upon the implementation.

Communication-aware techniques have been proposed in other related contexts. AMPI [20] can selectively schedule only threads whose MPI requests have completed, since it is built on Charm++ [25]. Charm++ provides a message-driven execution model, in which the arrival of messages triggers the execution of appropriate *chares*, the Charm++ word for a task. A similar technique was studied in [26], where a CPU core was used for progress and to partly control the continuation of OS-level threads by converting blocking calls to nonblocking calls. Further improvement can be made by a more tightly coupled design between the thread scheduler and the network interface [13]. In these works, however, executions require a centralized entity to control the executions of other entities and thus can result in wasting a CPU core for the dedicated scheduler.

VII. Conclusion

In this work, we tackled the problem of thread arbitration and synchronization in the context of MPI, and we proposed thread synchronization techniques to improve the communication performance in multithreaded communication scenarios using `MPI_THREAD_MULTIPLE`. Our techniques reduce the wasted time in the critical section while preserving data locality. Our method adopts a synchronization counter-based selective wakeup mechanism to reactivate waiting threads. It relies on electing and assigning at most one waiting thread to drive a communication context for improved data locality. Furthermore, active threads are prioritized and synchronized by using a locality-preserving lock that is hierarchical and exploits unbounded bias for high throughput. Our method does not count on an additional dedicated communication server but is incorporated into the implementation in a decentralized manner, which produces a scalable runtime systems. We implemented our techniques in a production MPI implementation, MPICH. Experimental results on multicore clusters show significant improvement in synthetic microbenchmarks and two MPI+OpenMP applications.

We plan to explore more communication kernels and applications with our runtime. When more and more clusters with a larger number of cores are deployed, we expect even greater performance improvements on those systems using our thread synchronization techniques.

Acknowledgment

We thank Marc Snir for a prior discussion that led to this project and paper. This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. The work also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *EuroMPI '08*, pp. 120–129.

[2] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded MPI communication on multicore petascale systems," in *EuroMPI '10*, pp. 11–20.

[3] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," in *PPoPP '12*, pp. 247–256.

[4] M. Chabbi and J. Mellor-Crummey, "Contention-conscious, locality-preserving locks," in *PPoPP '16*, pp. 22:1–22:14.

[5] D. Dice, "Malthusian locks," *CoRR*, vol. abs/1511.06035, 2015.

[6] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+threads: Runtime contention and remedies," in *PPoPP '15*, pp. 239–248.

[7] A. Amer, H. Lu, Y. Wei, J. Hammond, S. Matsuoka, and P. Balaji, "Locking aspects in multithreaded MPI implementations," Argonne National Lab., Tech. Rep. P6005-0516, 2016.

[8] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[9] T. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," University of Washington, Tech. Rep. TR 93-02-02, 1993.

[10] "MPICH: A high-performance and widely portable implementation of the MPI standard," http://www.mpich.org/.

[11] P. Dhabaleswar, "OSU Micro-Benchmarks 5.3," http://mvapich.cse.ohio-state.edu/benchmarks/.

[12] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[13] H. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *EuroMPI '16*.

[14] "TACC Stampede Cluster," http://www.xsede.org/resources/overview.

[15] "Graph 500," http://www.graph500.org/.

[16] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and hybrid MPI+Threads applications at scale: case study with BFS," in *PPMM '15*, 2015, pp. 1075–1083.

[17] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[18] F. Garcia, A. Calderón, and J. Carretero, "MiMPI: A multithred-safe implementation of MPI," in *EuroMPI '99*, pp. 207–214.

[19] A. Skjellum, B. Protopopov, and S. Hebert, "A thread taxonomy for MPI," in *MPIDC '96*.

[20] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *LCPC '03*, pp. 306–322.

[21] H. Tang, K. Shen, and T. Yang, "Compile/run-time support for threaded MPI execution on multiprogrammed shared memory machines," in *PPoPP '99*.

[22] E. D. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *HPCS '97*, pp. 156–163.

[23] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, and C. L. Mendes, "A new technique for data privatization in user-level threads and its use in parallel applications," in *SAC '10*.

[24] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Euro-Par '08*, pp. 78–88.

[25] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *OOPSLA '93*, pp. 91–108.

[26] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *SC '15*.